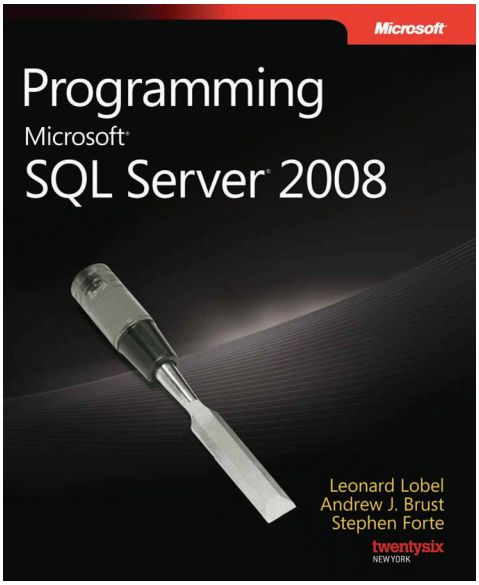

Programming Microsoft® SQL Server® 2008

Leonard Lobel,
Andrew J. Brust,
Stephen Forte
(twentysix new york)

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/12753.aspx

9780735625990

Microsoft® Press

Table of Contents

Chapter 8

Using FILESTREAM for Unstructured Data Storage

—Leonard Lobel

Applications today commonly work with unstructured data much more frequently than they did in the past. The accelerating data explosion of our times—relentlessly driven by dropping storage costs and rising storage capacities—continues to generate more and more unstructured data for us to handle. While applications of the past required little more than mapping data entry screens into rows and columns in the database and being able to perform fairly simple queries, today you also have data like audio, video, and other multimedia-type files to cope with. You might have to store employee photos, surveillance videos, recorded conversations, e-mail messages (including embedded attachments), trend analysis spreadsheets, content in proprietary formats, or other related artifacts with your database records. These unstructured types hold binary streams of information, commonly referred to as binary large object (BLOB) data. This BLOB data needs to be associated with the structured data that lives in your relational database.

Traditionally, there have been two solutions for combining structured table data with unstructured BLOB data in Microsoft SQL Server: either keep BLOBs in the database with all your structured relational data or store them outside the database (in either the file system or a dedicated BLOB store) with path references in the database that link to their external locations. Each of these strategies has pros and cons with respect to storage, performance, manageability, and programming complexity that we'll talk about—but neither of them is intrinsically native to the core database engine.

FILESTREAM is a major new feature in SQL Server 2008 that provides native support for efficiently handling BLOBs in the database. By improving the way we can now store and manage BLOB data, FILESTREAM offers a more efficient solution over traditional strategies. First we'll examine the pros and cons of storing BLOB data inside or outside the database, and then you'll learn how to enhance the storage and manipulation of BLOB data by leveraging FILESTREAM in SQL Server 2008.

BLOBs in the Database

Your first option, of course, is to store BLOB data directly in the table columns of your database. Do this by declaring a column as a *varbinary(max)* data type, which can store a single BLOB up to 2 gigabytes (GB) in size.

Important You should no longer use the *image* data type that was used to store BLOBs prior to SQL Server 2005. The *varbinary(max)* data type should now be used instead of *image*, which has been deprecated and may be removed from future versions of SQL Server.

Because BLOB data is stored inline with all the other structured table data, it is tightly integrated with the database. No effort is required on your part to link the relational data with its associated BLOB data. Management is therefore simplified, because everything is contained together within the file groups of a single database. Backup, restore, detach, copy, and attach operations on the database files encompass all structured and BLOB data together as a single entity. Transactional consistency is another important benefit of this approach. Because BLOB data is a physical part of the tables in the database, it is eligible to participate in transactions. If you begin a transaction, update some data, and then roll back the transaction, any BLOB data that was updated is also rolled back. Overall, the mixture of structured and BLOB data is handled quite seamlessly with this model.

Despite all these advantages, however, physically storing BLOBs in the database often results in a significant (and unacceptable) performance penalty. Because BLOB content (which tends to contain large amounts of data) is stored inline with structured data, it can consume a disproportionately large percentage of space in the database relative to the structured data. Query performance suffers greatly as a result, because the query processor needs to sift through much larger amounts of data in your tables that are holding inline BLOB content. The BLOBs also don't stream nearly as efficiently with *varbinary(max)* as they would if they were held externally in the file system or on a dedicated BLOB store. And last, *varbinary(max)* columns can store a maximum size of 2 GB. While this might not represent a limitation for handling typical documents, it can pose an obstacle for scenarios requiring much larger BLOB support (for example, where each row in a table of software products has a BLOB containing a distributable International Organization for Standardization [ISO] image of the software that can easily exceed 2 GB).

Note If you have modest storage requirements for BLOBs, where they are each typically 1 megabyte (MB) or smaller, you should consider keeping them in the database using the *varbinary(max)* data type instead of using the file system. Matters are simplified by storing the BLOBs inline with your tables rather than externally, and doing so will typically not affect performance when you are working with very few or very small BLOBs. Furthermore, you should consider caching small, frequently accessed BLOBs rather than repeatedly retrieving them from the database.

BLOBs in the File System

To address these performance bottlenecks, you can instead store BLOBs outside the database as ordinary files in the file system. With this approach, structured data in your relational tables merely contains path information to the unstructured BLOB data held in the file system. Applications use this path information as a link reference for tracking and locating the BLOB content associated with rows in the database tables. Because they are physically held in the file system, any BLOB can exceed 2 GB. In fact, their size is limited only by the host file system and available disk space. They also deliver much better streaming performance, since the file system provides a native environment optimized for streaming unstructured data, whereas the *varbinary(max)* column in the database does not. And because the physical database is much smaller without the BLOBs inside it, the query processor can continue to deliver optimal performance.

While physically separating structured and unstructured content this way does address the performance concerns of BLOBs, it also raises new issues because the data is now separated not only physically but logically as well. That is, SQL Server has absolutely no awareness of the association between data in the database and files stored externally in the file system that are referenced by path information in the database tables. Their coupling exists solely at the application level. Backup, restore, detach, copy, and attach operations on the database files therefore include only structured table data without any of the BLOB data that's in the file system. The integrated management benefits you get when storing BLOBs in the database are lost, and the administrative burden is increased by having to manage the file system separately.

Application development against this model is also more complex because of the extra effort required for linking between the database and the file system. The database offers no assistance in establishing and maintaining the references between its structured data and the external BLOBs, so it's up to the database designer and application developer to manage all of that on their own. And last, although perhaps most significant, there is no unified transactional control across both the database and the file system.

What's in an Attribute?

Of course, this discussion has been leading toward the new FILESTREAM feature, which combines the best of both worlds (and then some) in SQL Server 2008. First, to be clear, this is *not* technically a new data type in SQL Server. Instead, FILESTREAM is implemented as an *attribute* that you apply to the *varbinary(max)* data type. It might look innocent enough, but merely applying this attribute unleashes the FILESTREAM feature—an extremely efficient storage abstraction layer for managing unstructured data in the database. With this attribute applied, we continue to treat the *varbinary(max)* column *as though* its contents were stored

inline with our table data. Under the covers, however, the data is stored externally from the database in the server's NTFS file system.

With FILESTREAM, structured and unstructured data are logically connected but physically separated. The unstructured data is configured as just another file group in the database, so it participates in all logical database operations, including transactions and backup/restore. On disk, however, the BLOBs are stored as individual physical files in the NTFS file system that are created and managed automatically behind the scenes. SQL Server establishes and maintains the link references between the database and the file system. It knows about the unstructured BLOB data in the file system and considers the files holding BLOB data to be an integral part of the overall database. But the unstructured data doesn't impede query performance because it is not physically stored inline with table data. It's stored in the file system, which is highly optimized for streaming binary data. Logically, however, the database encompasses both the relational tables and the BLOB files in the file system. We therefore continue to treat BLOB data as though we were storing it inside the database itself, from both a development and an administrative perspective. For example, backing up the database includes all the BLOB data from the file system in the backup automatically.

Note Because the BLOB data is contained in its own database file group, you can easily exclude it from backups if desired or as needed.

The end result is that SQL Server 2008 uses the appropriate storage for structured and unstructured data—storing relational (structured) data in tables and BLOB (unstructured) data in files—in order to deliver the best possible performance all around. Because it does this completely transparently, we enjoy integrated management benefits over the database. The database engine handles the link references between the relational tables and their associated BLOB data in the file system for us. So we also enjoy simplified application development because we don't need to worry about the additional complexities of manually associating the database with the file system and keeping the two in sync, as we did in the past. Last, by leveraging the transactional capabilities of the NTFS file system, BLOB updates participate seamlessly with database transactions. If you're starting to get excited by all this, that's the idea! We're ready to dive in to some real code now that puts FILESTREAM to work for us.

Enabling FILESTREAM

Like many other features, FILESTREAM is disabled by default in SQL Server 2008, and you must first enable it before the feature can be used. Enabling FILESTREAM is slightly more involved than configuring other SQL Server features because it requires two distinct steps. First the feature needs to be enabled for the machine (Microsoft Windows service), and then it needs to be enabled for the server instance. These two FILESTREAM configuration layers

are by design, in order to draw a separation of security responsibilities between the roles of Windows administrator and SQL Server administrator.

Enabling FILESTREAM for the Machine

The first step is to enable FILESTREAM for the machine by setting an access level. This step can actually be performed at the time that SQL Server is initially installed by choosing a FILESTREAM access level during setup. The default access level, as already mentioned, is disabled. To enable FILESTREAM for the machine after SQL Server has been installed, the Windows administrator uses the SQL Server Configuration Manager to set the access level. (This tool can be launched from the Configuration Tools folder of the Microsoft SQL Server 2008 program group on the Start menu.)

The SQL Server Configuration Manager opens with a list of services displayed in the main panel. In the list of services, right-click the SQL Server instance that you want to enable FILESTREAM for, and then choose Properties. In the Properties dialog box, select the FILESTREAM tab. The three check boxes on the FILESTREAM tab allow you to select the various levels of FILESTREAM access. Figure 8-1 shows the Properties dialog box with all three check boxes selected.

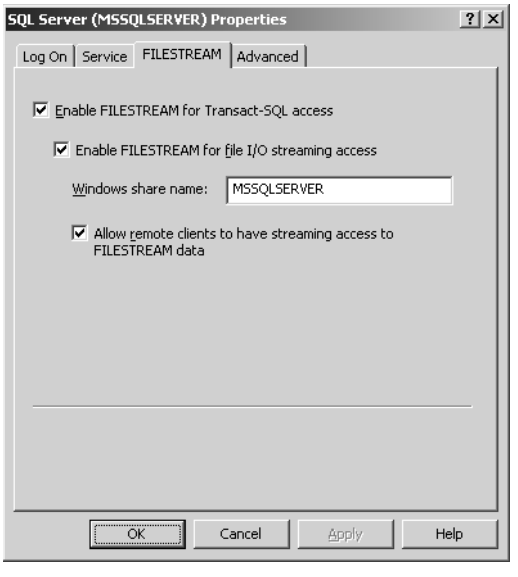

FIGURE 8-1 Enabling FILESTREAM for the machine to support file I/O streaming access by remote clients

When all three check boxes are cleared, FILESTREAM is completely disabled. Selecting the first check box enables FILESTREAM, but only for Transact-SQL (T-SQL) access. This provides a completely transparent FILESTREAM implementation, but it doesn't let you take advantage of streamed file access between the database and your client applications.

The real power of FILESTREAM comes into play when you enable direct file I/O streaming, which delivers the best possible performance for accessing BLOB data in the file system with SQL Server. You enable direct file I/O streaming access by selecting the second check box. Streamed file access also creates a Windows share name that is used to construct logical Universal Naming Convention (UNC) paths to BLOB data during FILESTREAM access, as we'll see further on when we use the *OpenSqlFilestream* function in our sample .NET applications. The share name is specified in a text box after the second check box and is set by default to the same name as the server instance (*MSSQLSERVER*, in this example).

In most cases, client applications will not be running on the same machine as SQL Server, so you will usually also need to select the third check box to enable FILESTREAM for remote client file I/O streaming access. One exception to this general practice might be when using Microsoft SQL Server 2008 Express edition as a local data store for a client application with everything running on the same machine. In this case, you would use the more secure setting and leave the third check box cleared. Doing so would enable file I/O streaming access for the local client application but deny such access to remote clients.

Throughout the rest of this chapter, we'll be building several sample .NET applications that work with FILESTREAM. These applications will demonstrate how to use *OpenSqlFilestream* for file I/O streaming access, so at least the first two check boxes must be selected for the sample code to work. If you are running the applications on a different machine than SQL Server, you will also need to select the third check box to allow remote access.

More Info There is no T-SQL equivalent script that can set the FILESTREAM access level for the machine. However, Microsoft posts a VBScript file available over the Internet that allows you to enable FILESTREAM from the command line as an alternative to using SQL Server Configuration Manager. At press time, the download page for this script is *http://www.codeplex.com/SQLSrvEngine/Wiki/View.aspx?title=FileStreamEnable&referringTitle=Home*. An Internet shortcut to this URL is included with this chapter's sample code on the book's companion Web site. Alternatively, try running a Web search on "How to enable FILESTREAM from the command line."

Enabling FILESTREAM for the Server Instance

The second step is for the SQL Server administrator to enable FILESTREAM for the server instance. The concept here is similar to the first step in that varying levels of access are defined. FILESTREAM can be enabled for the server instance with a simple call to the *sp_configure* system stored procedure, as follows:

```
EXEC sp_configure filestream_access_level, n
RECONFIGURE
```

In the preceding code, replace *n* with a number from 0 to 2 to set the access level. The value *0* disables the FILESTREAM feature completely. Setting the access level to *1* enables

FILESTREAM for T-SQL access only, and setting it to *2* enables FILESTREAM for full access (which includes local or remote file I/O streaming access as enabled for the machine in the first step). To support our sample .NET applications that will demonstrate file I/O streaming access using *OpenSqlFilestream*, you'll need to select level 2 (full access).

Note Naturally, the access level defined for the server instance must be supported by the access level defined for the machine. Typically, therefore, the access levels between the machine and the server instance should be set to match each other.

You can also set the FILESTREAM access level for the server instance in SQL Server Management Studio from the Advanced Server Properties dialog box. Right-click any server instance in Object Explorer, choose Properties, and then select the Advanced page. The various levels are available as choices in the Filestream Access Level drop-down list, as shown in Figure 8-2.

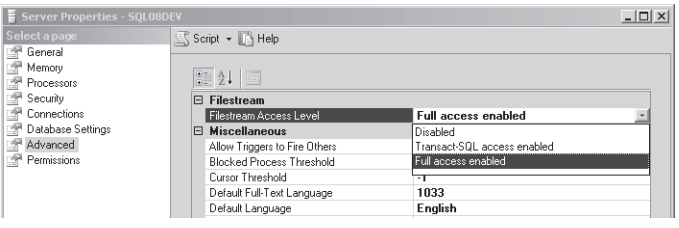

FIGURE 8-2 Selecting the FILESTREAM configuration level in SQL Server Management Studio

Creating a FILESTREAM-Enabled Database

Once FILESTREAM is enabled for both the machine and the server instance, any database running on the server instance can support unstructured data by defining a file group with the new *FILEGROUP…CONTAINS FILESTREAM* clause of the *CREATE DATABASE* statement. For example, the statement in Listing 8-1 creates a *PhotoLibrary* database that can store pictures using FILESTREAM.

LISTING 8-1 Creating a FILESTREAM-enabled catabase with *FILEGROUP…CONTAINS FILESTREAM*

```
CREATE DATABASE PhotoLibrary
 ON PRIMARY
  (NAME = PhotoLibrary_data,
   FILENAME = 'C:\PhotoLibrary\PhotoLibrary_data.mdf'),
 FILEGROUP FileStreamGroup1 CONTAINS FILESTREAM
  (NAME = PhotoLibrary_group2,
   FILENAME = 'C:\PhotoLibrary\Photos')
 LOG ON
  (NAME = PhotoLibrary_log,
   FILENAME = 'C:\PhotoLibrary\PhotoLibrary_log.ldf')
```

The *FILEGROUP…CONTAINS FILESTREAM* clause in this otherwise ordinary *CREATE DATABASE* statement enables the FILESTREAM feature for the *PhotoLibrary* database.

A few simple but important things warrant mention at this point. To begin, as when creating any database, the directory (or directories) specified for the primary and log file groups must exist at the time the database is created. In our example, the C:\PhotoLibrary directory specified by *FILENAME* in the *ON PRIMARY* and *LOG ON* clauses must exist, or the *CREATE DATABASE* statement will fail.

Interestingly, and somewhat oddly, the *FILENAME* specified in the new *FILEGROUP… CONTAINS FILESTREAM* clause does not actually specify the name of a file but instead specifies the name of a directory. And unlike the primary and log file group directories, this directory must *not* exist at the time that the database is created (although the path leading up to the final directory must exist), or the *CREATE DATABASE* statement will fail as well. Instead, SQL Server takes control of creating and managing this directory, much as it does for creating and managing the .mdf and .ldf files in the other file groups. In our example, SQL Server will automatically create the C:\PhotoLibrary\Photos folder when the *CREATE DATABASE* statement is executed and will then use that folder for storing all BLOB data—photos, in our example—in the *PhotoLibrary* database.

When we execute this *CREATE DATABASE* statement with an empty C:\PhotoLibrary directory, SQL Server creates the usual .mdf and .ldf files for us and also creates the Photos subdirectory for the FILESTREAM group, as shown in Figure 8-3.

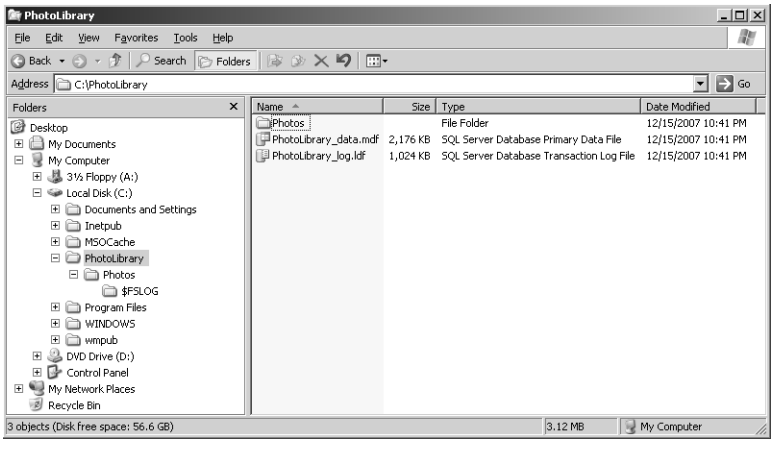

FIGURE 8-3 FILESTREAM storage in the file system

Behind the scenes, SQL Server will store all our pictures as files in the Photos subdirectory and track the references between those picture files and the relational tables that they logically belong to in database columns defined as *varbinary(max) FILESTREAM*. Unless we explicitly exclude the *FileStreamGroup1* file group from a backup or restore command, all our

picture files in the *Photos* subdirectory will be included with the relational database in the backup or restore operation.

Creating a Table with FILESTREAM Columns

We're now ready to create the *PhotoAlbum* table. SQL Server requires that any table using FILESTREAM storage have a *uniqueidentifier* column that is not nullable and that specifies the *ROWGUIDCOL* attribute. You must also create a unique constraint on this column. Only one *ROWGUIDCOL* column can be defined in any given table, although defining one then allows you to declare any number of *varbinary(max) FILESTREAM* columns in the table that you want for storing BLOB data. The statement in Listing 8-2 creates the *PhotoAlbum* table with a *Photo* column declared as *varbinary(max) FILESTREAM*.

LISTING 8-2 Creating a FILESTREAM-enabled table

```
CREATE TABLE PhotoAlbum(
 PhotoId int PRIMARY KEY,
 RowId uniqueidentifier ROWGUIDCOL NOT NULL UNIQUE DEFAULT NEWSEQUENTIALID(),
 Description varchar(max),
 Photo varbinary(max) FILESTREAM DEFAULT(0x))
```

With this statement, we satisfy the FILESTREAM requirement for the *ROWGUIDCOL* column, yet we won't actually have to do anything to maintain that column. By declaring the *RowId* column with its *DEFAULT* value set to call the *NEWSEQUENTIALID* function, we can just pretend this column doesn't even exist—simply not providing values for it will cause SQL Server to automatically generate an arbitrary globally unique identifier (GUID) for the column that it needs to support FILESTREAM on the table. The column is set to not accept *NULL* values and is defined with the required unique constraint.

We have also declared an integer *PhotoId* column for the table's primary key value. We'll use the *PhotoId* column to identify individual photos in the album, and SQL Server will use the *RowId* column to track and cross-reference photos in the file system with rows in the *PhotoAlbum* table. The *Photo* column holds the actual BLOB itself, being defined as a *varbinary(max)* data type with the *FILESTREAM* attribute applied. This means that it gets treated like a regular *varbinary(max)* column, but we know that its BLOB is really being stored in the file system by SQL Server internally. For now, just take note that we've defined a default 0x binary value for the *Photo* column. This will come into play when we start streaming content with client code, but we're not there yet.

Manipulating BLOB data is not something that is easily or practically done in T-SQL. Of course, you can specify small binary streams inline directly in your T-SQL code, or embed and extract binary streams using byte arrays as you could with an ordinary *varbinary(max)* column. But the proper (and fastest) way to get data into and out of FILESTREAM columns is by

using a native or managed client that calls the *OpenSqlFilestream* function provided by the SQL Server 2008 native client API.

With *OpenSqlFilestream*, native or managed code applications can use either the *ReadFile* and *WriteFile* Microsoft Win32 application programming interface (API) functions or the .NET *FileStream* class to deliver high-performance streaming of BLOB data. In the next section, you'll see exactly how to use the managed *FileStream* class in C# for storing and retrieving pictures in the *Photo* column. But right now, we're going to do something rather contrived instead and use T-SQL to cast string data into and out of the *varbinary(max)* data type in the *Photo* column. We're doing this so that you can come to understand FILESTREAM one step at a time, and the first thing we want to do is observe the effects on the NTFS file system as SQL Server uses it to store BLOB data in *varbinary(max) FILESTREAM* columns. So we'll begin modestly with the following *INSERT* statement that adds our first row to the *PhotoAlbum* table:

```
INSERT INTO PhotoAlbum(PhotoId, Description, Photo)
 VALUES(1, 'First pic', CAST('BLOB' As varbinary(max))
```

This *INSERT* statement reads no differently than it would if we were using a regular *varbinary(max)* column for the *Photo* column without the *FILESTREAM* attribute. It appears to store the unstructured *Photo* column data inline with the rest of the relational columns, and it appears the same way when returning the data back with a *SELECT* query, as shown here:

```
SELECT *, CAST(Photo AS varchar) AS PhotoText FROM PhotoAlbum
GO

PhotoId RowId                                Description Photo      PhotoText
------- ------------------------------------ ----------- ---------- ---------
1       c04a7930-89ab-dc11-91e3-0003ff399330 First pic   0x424C4F42 BLOB

(1 row(s) affected)
```

However, if we peek beneath the covers, we can see that SQL Server is actually storing the *Photo* column outside the database in the file system. Because everything is tracked internally for us, we don't really need to understand the precise manner in which files and folders are named, organized, and cross-referenced back to the relational database. But just by drilling down and probing the subfolders beneath the *Photos* directory, we discover that there is in fact a new file stored in the file system created as a result of the *INSERT* statement that we can view by right-clicking the file name and then choosing Open, as shown in Figure 8-4.

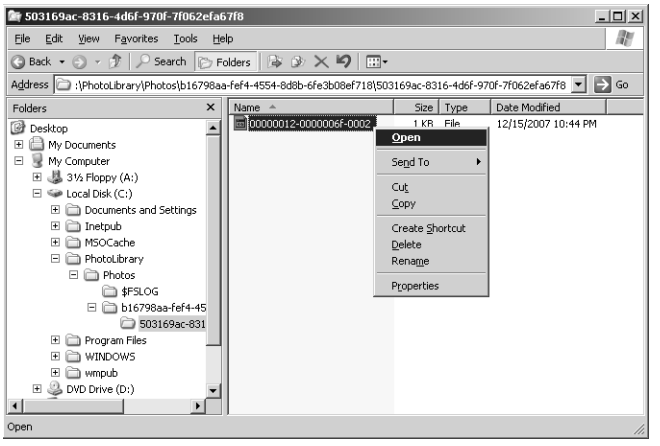

FIGURE 8-4 Exploring the FILESTREAM file system

If we select Notepad to open the file, we get proof positive that the unstructured content of the *Photo* column is stored outside the database and in the file system. In this example, the text *BLOB* that was inserted into the *Photo* column is stored in the file that we've just opened in Notepad, as shown in Figure 8-5.

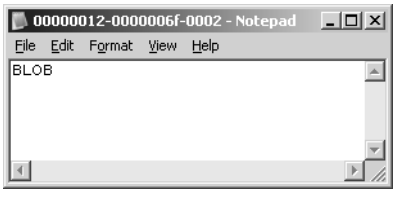

FIGURE 8-5 Examining unstructured FILESTREAM content in Notepad

This clearly demonstrates how FILESTREAM data is logically connected to—but physically separated from—the database. Because the unstructured data is stored entirely in the file system, we can easily alter its content by directly updating the file itself in Notepad without even involving the database. To prove the point further, let's change the text in the file from *BLOB* to *Cool* and save the changes back to the file system, as shown in Figure 8-6.

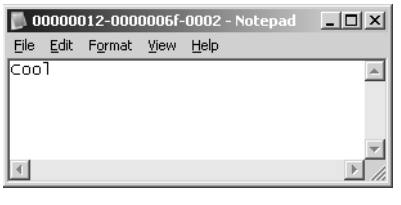

FIGURE 8-6 Changing FILESTREAM content directly in the file system

The changed FILESTREAM data is reflected in the same *SELECT* statement we ran earlier, as shown here:

```
SELECT *, CAST(Photo AS varchar) AS PhotoText FROM PhotoAlbum
GO

PhotoId RowId                                Description Photo      PhotoText
------- ------------------------------------ ----------- ---------- ---------
1       c04a7930-89ab-dc11-91e3-0003ff399330 First pic   0x436F6F6C Cool

(1 row(s) affected)
```

Important We performed this exercise to demonstrate and verify that the file system is being used to store FILESTREAM data. Having said that, you should *never* tamper directly with files in the file system this way. With respect to FILESTREAM, consider the file system as part of the database file groups (.mdf and .ldf files); it gets managed by SQL Server exclusively.

The *OpenSqlFilestream* Native Client API

With an understanding and appreciation of how FILESTREAM is implemented internally by SQL Server, we're ready now to move forward and store real binary picture data in the *Photo* column. As we mentioned earlier, this is best achieved by writing client code that calls the *OpenSqlFilestream* function provided by the SQL Server native client API.

When you work with *OpenSqlFilestream*, you always work with transactions (even for read access). There is no way to avoid them, since FILESTREAM by design coordinates transactional integrity across structured and unstructured data access between SQL Server and the NTFS file system. (However, we should stress that you normally should try not to read while in a transaction when you're not working with *OpenSqlFilestream*.)

Here's how it works. We first start an ordinary database transaction, after which we perform any number of normal data manipulation language (DML) operations (such as inserts or updates) on the database. When we access a *varbinary(max) FILESTREAM* column, SQL Server automatically initiates an NTFS file system transaction and associates it with the database transaction. SQL Server also ensures that both the database transaction and the file system transaction will either commit or roll back together.

To then stream BLOBs in and out, there are two key pieces of information we need to obtain. First, we need the file system transaction context, which is returned by the *GET_FILESTREAM_TRANSACTION_CONTEXT* function. (This function returns *NULL* if a transaction has not yet been established.) Second, we need a logical UNC path to the file holding the BLOB on the server, which is returned by the *PathName* method invoked on a *varbinary(max) FILESTREAM* value instance. These two pieces of information are then passed as inputs to the *OpenSqlFilestream* function, which returns a file handle back to us that we can use to perform

efficient streaming I/O operations directly against the BLOB data stored in the file system on the server. Only when the database transaction is committed does SQL Server permanently save changes both to the database (from the DML operations) and to the file system (from the streaming I/O operations). Similarly, rolling back the transaction undoes changes to both the database and the file system.

Note The UNC reference returned by the *PathName* method is *not* a real path to the physical file system on the server. Rather, *PathName* returns a fabricated path to be used by *OpenSqlFilestream* to enable direct streaming between the file system and client applications. (The share name in this UNC path is based on the share name specified when FILESTREAM was enabled for the machine, as described earlier in this chapter.) The file system itself is secured on the server no differently than the data and transaction file groups (.mdf and .ldf files) are secured. Users should never be granted direct access to the file system on the server. Normal SQL Server column-level security permissions apply to *varbinary(max) FILESTREAM* columns.

The handle returned by *OpenSqlFilestream* can be used with the Win32 *ReadFile* and *WriteFile* API functions for client applications written in native code, such as C++. The handle can also be used by the *FileStream* class in .NET for client applications written in managed code, such as C# or Visual Basic .NET. Continuing with our photo library example, we'll proceed to create a Windows Forms application in C# that implements all of the key pieces that bring a FILESTREAM application together—nothing more, nothing less. Our application will allow the user to create a new photo in the database that streams the BLOB into the *Photo* column and to select a photo that streams the BLOB back out from the *Photo* column into a *PictureBox* control for display. *OpenSqlFilestream* will provide us with the handle we need to read and write the binary picture data using the ordinary *FileStream* class defined in the *System.IO* namespace.

File-Streaming in .NET

We'll begin with the Windows user interface (UI), which is very simple. Start Visual Studio 2008, and then create a new C# Windows Forms application. Design a form with two separate group boxes: one at the top of the form for inserting photos and another beneath it for selecting photos. Provide labels and text boxes for entering a photo ID, file name, and description in the top group box, along with a link label to invoke a save operation. In the bottom group box, provide a text box and label for entering a photo ID and a link label to invoke a load operation. Include a label to display the description returned from the database and a picture box to display the photo BLOB returned via FILESTREAM. After performing some aesthetic alignment and formatting, your form should appear something like the one shown in Figure 8-7.

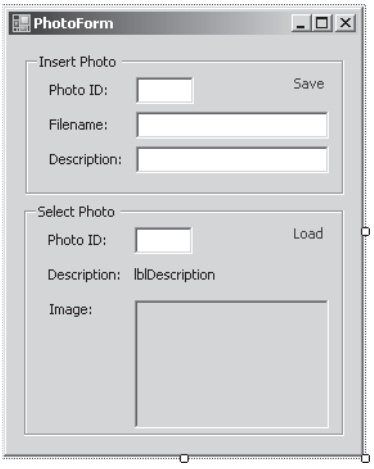

FIGURE 8-7 Simple FILESTREAM Windows UI form

We'll write only a very small amount of code behind this Windows form, and we'll implement the FILESTREAM logic in a separate data access class that can be reused across a variety of user interface technologies, including Windows Forms, ASP.NET, and Windows Presentation Foundation (WPF). Let's add the code behind the click events for this form's Save and Load link labels that hooks into the data access class named *PhotoData* (which we'll create right after the UI), as shown in Listing 8-3.

LISTING 8-3 UI calls into FILESTREAM data access class for saving and loading image files

```
private void lnkSave_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e)
{
    int photoId = int.Parse(this.txtSavePhotoId.Text);
    string desc = this.txtDescription.Text;
    string filename = this.txtFilename.Text;

    PhotoData.InsertPhoto(photoId, desc, filename);
}

private void lnkLoad_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e)
{
    int photoId = int.Parse(this.txtLoadPhotoId.Text);

    string desc;
    Image photo = PhotoData.SelectPhoto(photoId, out desc);

    this.lblDescription.Text = desc;
    this.picImage.Image = photo;
}
```

When the user clicks *Save*, the code retrieves the new photo ID, description, and file name from the three text boxes and passes them to the *InsertPhoto* method of the *PhotoData* class.

When the user specifies a photo ID and clicks Load, the code calls the *SelectPhoto* method of the *PhotoData* class to retrieve the requested description and image for display.

Understanding FILESTREAM Data Access

All the magic happens inside the *PhotoData* class, which is a UI-agnostic data access class. This design draws a clear separation between data access and the UI, with only a minimal amount of UI-specific code to maintain. Listing 8-4 shows the complete source code for the *PhotoData* class.

Note The *PhotoData* class takes a minimalist approach for proof-of-concept purposes only. The connection string is defined as a hard-coded constant; a real-world application should encrypt and store the connection string elsewhere (such as a configuration settings file). The code also employs the *using* construct in C# to ensure that all objects that allocate unmanaged resources such as database connections and file handles are disposed of properly even if an exception occurs, without including any additional error handling logic. Once again, real-world applications should implement a robust and reliable exception handling strategy that includes the use of *try/catch/finally* blocks, error logging, and validation.

LISTING 8-4 Implementing a FILESTREAM data access managed client class

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Drawing;
using System.IO;

using Microsoft.Win32.SafeHandles;

namespace PhotoLibraryApp
{
  public class PhotoData
  {
    private const string ConnStr =
      "Data Source=.;Integrated Security=True;Initial Catalog=PhotoLibrary;";

    #region "Insert Photo"

    public static void InsertPhoto(int photoId, string desc, string filename)
    {
      const string InsertCmd =
        "INSERT INTO PhotoAlbum(PhotoId, Description)" +
        " VALUES(@PhotoId, @Description)";

      using (SqlConnection conn = new SqlConnection(ConnStr))
      {
        conn.Open();
```

```
      using (SqlTransaction txn = conn.BeginTransaction())
      {
        using (SqlCommand cmd = new SqlCommand(InsertCmd, conn, txn))
        {
          cmd.Parameters.Add("@PhotoId", SqlDbType.Int).Value = photoId;
          cmd.Parameters.Add("@Description", SqlDbType.VarChar).Value = desc;
          cmd.ExecuteNonQuery();
        }

        SavePhotoFile(photoId, filename, txn);
        txn.Commit();
      }

      conn.Close();
    }
  }

  private static void SavePhotoFile
   (int photoId, string filename, SqlTransaction txn)
  {
    const int BlockSize = 1024 * 512;

    FileStream source = new FileStream(filename, FileMode.Open, FileAccess.Read);

    SafeFileHandle handle = GetOutputFileHandle(photoId, txn);
    using (FileStream dest = new FileStream(handle, FileAccess.Write))
    {
      byte[] buffer = new byte[BlockSize];
      int bytesRead;
      while ((bytesRead = source.Read(buffer, 0, buffer.Length)) > 0)
      {
        dest.Write(buffer, 0, bytesRead);
        dest.Flush();
      }
      dest.Close();
    }

    source.Close();
  }

  private static SafeFileHandle GetOutputFileHandle
   (int photoId, SqlTransaction txn)
  {
    const string GetOutputFileInfoCmd =
      "SELECT Photo.PathName(), GET_FILESTREAM_TRANSACTION_CONTEXT()" +
      " FROM PhotoAlbum" +
      " WHERE PhotoId = @PhotoId";

    SqlCommand cmd = new SqlCommand(GetOutputFileInfoCmd, txn.Connection, txn);
    cmd.Parameters.Add("@PhotoId", SqlDbType.Int).Value = photoId;

    string filePath;
    byte[] txnToken;
```

```
  using (SqlDataReader rdr = cmd.ExecuteReader(CommandBehavior.SingleRow))
  {
    rdr.Read();
    filePath = rdr.GetSqlString(0).Value;
    txnToken = rdr.GetSqlBinary(1).Value;
    rdr.Close();
  }

  SafeFileHandle handle =
    NativeSqlClient.GetSqlFilestreamHandle
     (filePath, NativeSqlClient.DesiredAccess.ReadWrite, txnToken);

  return handle;
}

#endregion

#region "Select Photo"

public static Image SelectPhoto(int photoId, out string desc)
{
  const string SelectCmd =
  "SELECT Description, Photo.PathName(), GET_FILESTREAM_TRANSACTION_CONTEXT()" +
  " FROM PhotoAlbum" +
  " WHERE PhotoId = @PhotoId";

  Image photo;

  using (SqlConnection conn = new SqlConnection(ConnStr))
  {
    conn.Open();

    using (SqlTransaction txn = conn.BeginTransaction())
    {
      string filePath;
      byte[] txnToken;

      using (SqlCommand cmd = new SqlCommand(SelectCmd, conn, txn))
      {
        cmd.Parameters.Add("@PhotoId", SqlDbType.Int).Value = photoId;

        using (SqlDataReader rdr = cmd.ExecuteReader(CommandBehavior.SingleRow))
        {
          rdr.Read();
          desc = rdr.GetSqlString(0).Value;
          filePath = rdr.GetSqlString(1).Value;
          txnToken = rdr.GetSqlBinary(2).Value;
          rdr.Close();
        }
      }
```

```
            photo = LoadPhotoImage(filePath, txnToken);

            txn.Commit();
          }

          conn.Close();
        }

        return photo;
      }

      private static Image LoadPhotoImage(string filePath, byte[] txnToken)
      {
        Image photo;

        SafeFileHandle handle =
          NativeSqlClient.GetSqlFilestreamHandle
           (filePath, NativeSqlClient.DesiredAccess.Read, txnToken);

        using (FileStream fs = new FileStream(handle, FileAccess.Read))
        {
          photo = Image.FromStream(fs);
          fs.Close();
        }

        return photo;
      }

      #endregion
    }
  }
```

There is also a small source file in our application named *NativeSqlClient* that encapsulates the Component Object Model (COM) Interop and native code call interface details for invoking *OpenSqlFilestream* from our managed code. It's this *NativeSqlClient* class that actually calls *OpenSqlFilestream*, whereas our managed code client applications call into *NativeSqlClient* for issuing all *OpenSqlFilestream* requests. We'll begin our in-depth code coverage with the *PhotoData* class and then look at the supporting *NativeSqlClient* class at the point that we call into it.

We'll start at the top with some required namespace inclusions. The one to take notice of is *Microsoft.Win32.SafeHandles*, which defines the *SafeFileHandle* object returned by *OpenSqlFilestream* that we'll be using to stream BLOBs. (No special assembly reference is required to use the *Microsoft.Win32.SafeHandles.SafeFileHandle* class, because it is provided by the core .NET library assembly mscorlib.dll.) We also define a connection string as a hard-coded constant, which of course is for demonstration purposes only. A real-world application would encrypt and store the connection string elsewhere (such as a configuration settings file), but we're keeping our example simple.

The first method defined in the class is *InsertPhoto*, which accepts a new photo integer ID, string description, and full path to an image file to be saved to the database, as shown here:

```
public static void InsertPhoto(int photoId, string desc, string filename)
{
    const string InsertCmd =
        "INSERT INTO PhotoAlbum(PhotoId, Description)" +
        " VALUES(@PhotoId, @Description)";

    using(SqlConnection conn = new SqlConnection(ConnStr))
    {
        conn.Open();

        using(SqlTransaction txn = conn.BeginTransaction())
        {
            using(SqlCommand cmd = new SqlCommand(InsertCmd, conn, txn))
            {
                cmd.Parameters.Add("@PhotoId", SqlDbType.Int).Value = photoId;
                cmd.Parameters.Add("@Description", SqlDbType.VarChar).Value = desc;
                cmd.ExecuteNonQuery();
            }

            SavePhotoFile(photoId, filename, txn);
            txn.Commit();
        }

        conn.Close();
    }
}
```

The method first creates and opens a new *SqlConnection* and then initiates a database transaction using the *SqlTransaction* class against the open connection. Next it creates a *SqlCommand* object associated with the open connection and initiated transaction, and prepares its command text with an *INSERT* statement (defined in the *InsertCmd* string constant) that stores the photo ID and description values in a new *PhotoAlbum* record. Our *INSERT* statement does not provide a value for *RowId* and instead allows SQL Server to automatically generate and assign a new *uniqueidentifier ROWGUID* value by default just as before, when we used T-SQL to insert the first row. We also do not provide a value for the *Photo* column—and now is exactly when the default 0x value that we defined earlier for the *Photo* column comes into play. After executing the *INSERT* by invoking *ExecuteNonQuery*, the transaction is still pending. Although the row has been added, it will roll back (disappear) if a problem occurs before the transaction is committed. Because we didn't provide a BLOB value for the *Photo* column in the new row, SQL Server honors the default value 0x that we established for it in the *CREATE TABLE* statement for *PhotoAlbum*. Being a *varbinary(max)* column decorated with the *FILESTREAM* attribute, this results in an empty file being added to the file system that is linked to the new row. And like the new row, this new empty BLOB file will disappear if the database transaction does not commit successfully.

Important You cannot open a file handle to a *NULL* column value. If you want to use *OpenSqlFilestream*, a binary 0x value should always be used with *varbinary(max) FILESTREAM* columns when inserting new rows. This will result in the creation of a zero-length file that can be streamed to (overwritten) by calling *OpenSqlFilestream*, as we're doing now.

It is precisely at this point that we call the *SavePhotoFile* method to stream the specified image file into the *Photo* column of the newly inserted *PhotoAlbum* row, overwriting the empty file just added by default. When control returns from *SavePhotoFile*, the transaction is finally committed and the connection is closed. This permanently updates both the database and the file system with the structured and unstructured content for a new *PhotoAlbum* row.

The *SavePhotoFile* method reads from the source file and writes to the database FILESTREAM storage in 512-KB chunks using an ordinary *FileStream* object, as shown here:

```
private static void SavePhotoFile(int photoId, string filename, SqlTransaction txn)
{
    const int BlockSize = 1024 * 512;

    FileStream source = new FileStream(filename, FileMode.Open, FileAccess.Read);

    SafeFileHandle handle = GetOutputFileHandle(photoId, txn);
    using(FileStream dest = new FileStream(handle, FileAccess.Write))
    {
        byte[] buffer = new byte[BlockSize];
        int bytesRead;
        while((bytesRead = source.Read(buffer, 0, buffer.Length)) > 0)
        {
            dest.Write(buffer, 0, bytesRead);
            dest.Flush();
        }
        dest.Close();
    }

    source.Close();
}
```

The method begins by defining a *BlockSize* integer constant that is set to a reasonable value of 512 KB. Picture files larger than this will be streamed to the server in 512-KB pieces. The local source image file is first opened on a read-only *FileStream*. In order to obtain a writable *FileStream* on the output file in SQL Server, we call the *GetOutputFileHandle* method, passing in the photo ID and pending transaction and receiving back a *SafeFileHandle* object (defined in the *Microsoft.Win32.SafeHandles* namespace imported with a *using* statement at the top of the source file). The *FileStream* class offers a constructor that accepts a *SafeFileHandle* object, which we use to gain write access to the destination BLOB on the database server's NTFS file system. Remember that this output file is enlisted in an NTFS transaction and will not be permanently saved until the database transaction is committed by the code that is calling *SavePhotoFile*.

The rest of the *SavePhotoFile* method implements a simple loop that reads from the source *FileStream* and writes to the destination *FileStream* until the entire source file is processed and then closes both streams.

Let's now examine the *GetOutputFileHandle* method, which is called by *SavePhotoFile* to obtain the destination handle for streaming to the BLOB file:

```
private static SafeFileHandle GetOutputFileHandle(int photoId, SqlTransaction txn)
{
    const string GetOutputFileInfoCmd =
        "SELECT GET_FILESTREAM_TRANSACTION_CONTEXT(), Photo.PathName()" +
        " FROM PhotoAlbum" +
        " WHERE PhotoId = @PhotoId";

    SqlCommand cmd = new SqlCommand(GetOutputFileInfoCmd, txn.Connection, txn);
    cmd.Parameters.Add("@PhotoId", SqlDbType.Int).Value = photoId;

    string filePath;
    byte[] txnToken;

    using(SqlDataReader rdr = cmd.ExecuteReader(CommandBehavior.SingleRow))
    {
        rdr.Read();
        txnToken = rdr.GetSqlBinary(0).Value;
        filePath = rdr.GetSqlString(1).Value;
        rdr.Close();
    }

    SafeFileHandle handle =
        NativeSqlClient.GetSqlFilestreamHandle
        (filePath, NativeSqlClient.DesiredAccess.ReadWrite, txnToken);

    return handle;
}
```

This code is the key to using FILESTREAM. To reiterate, it is called at a point in time after a new row has been added to the *PhotoAlbum* table and a new, empty related BLOB file has been added to the file system but before the transactions that those actions are enlisted on have been committed. This is precisely the time for us to hook into the process and stream BLOB data into the database using *OpenSqlFilestream*. Recall from our discussion earlier that in order to do that, we need two pieces of information: a transactional context token and a logical UNC path name to the file itself. We therefore obtain both these items in a single-row *SqlDataReader* using a *SELECT* statement that returns *GET_FILESTREAM_TRANSACTION_CONTEXT* and *Photo.PathName*.

Because we began the database transaction before running the *INSERT* statement, SQL Server initiated a file system transaction in NTFS over the FILESTREAM data in the new row's *Photo* column. The *GET_FILESTREAM_TRANSACTION_CONTEXT* function returns a handle to that NTFS transaction. SQL Server will automatically commit this NTFS transaction when

we commit the database transaction or roll back the NTFS transaction if we roll back the database transaction. When we obtain the transaction context, which is a *SqlBinary* value, we store it in a byte array named *txnToken*.

The second value returned by our *SELECT* statement is *Photo.PathName*, which returns a fabricated path (in UNC format, including the file name) to the BLOB for the selected *PhotoId*. What we're essentially doing with the *WHERE* clause is reading back the same row we have just added (but not yet committed) to the *PhotoAlbum* table in order to get the full path name to the BLOB stored in the new file that was just created (also not yet committed) in the file system. We're then storing it in a string variable named *filePath*.

Armed with both the FILESTREAM transaction context and the full path name to the BLOB file, we have what we need to call the native *OpenSqlFilestream* SQL client function and obtain a handle to the output file for streaming our content. However, we don't actually call *OpenSqlFilestream* directly from our data access class (although we certainly could). Instead, we call *GetSqlFilestreamHandle*, defined in our supporting *NativeSqlClient* class, which in turn calls *OpenSqlFilestream*, as shown in Listing 8-5.

LISTING 8-5 Calling *OpenSqlFilestream*

```
using System;
using System.Runtime.InteropServices;

using Microsoft.Win32.SafeHandles;

namespace PhotoLibraryFilestreamDemo
{
    public class NativeSqlClient
    {
        public enum DesiredAccess : uint
        {
            Read,
            Write,
            ReadWrite,
        }

        [DllImport("sqlncli10.dll", SetLastError = true, CharSet = CharSet.Unicode)]
        private static extern SafeFileHandle OpenSqlFilestream(
            string path,
            uint access,
            uint options,
            byte[] txnToken,
            uint txnTokenLength,
            Sql64 allocationSize);
```

```
        [StructLayout(LayoutKind.Sequential)]
        private struct Sql64
        {
            public Int64 QuadPart;
            public Sql64(Int64 quadPart)
            {
                this.QuadPart = quadPart;
            }
        }

        public static SafeFileHandle GetSqlFilestreamHandle
         (string filePath, DesiredAccess access, byte[] txnToken)
        {
            SafeFileHandle handle = OpenSqlFilestream(
                filePath,
                (uint)access,
                0,
                txnToken,
                (uint)txnToken.Length,
                new Sql64(0));

            return handle;
        }
    }
}
```

As you can see, the *GetSqlFilestreamHandle* method merely wraps the native *OpenSqlFilestream* function, which is defined with an external reference to *sqlncli10.dll* (SQL Native Client version 10) by using the *DllImport* attribute. *GetSqlFilestreamHandle* accepts the transaction context token and the full path to the BLOB file obtained by the *GET_FILESTREAM_TRANSACTION_CONTEXT* function and the *PathName* method. It also accepts an enumeration value that specifies the desired access mode, which can be *Read*, *Write*, or *ReadWrite*. The *OpenSqlFilestream* function requires other parameters that are not generally applicable for standard FILESTREAM usage, such as the unsigned 32-bit options and 64-bit allocation size arguments. These simply get passed in as 0.

Tip Of course, *PhotoData* could have called *OpenSqlFilestream* directly. The purpose of our *NativeSqlClient* client class is to keep the COM Interop and 64-bit SQL integers out of our data access code, drawing a separation between managed and native code concerns. The result is neater and more maintainable code. We are exposing a simple *GetSqlFilestreamHandle* managed code method wrapper around the native *OpenSqlFilestream* function, so our *PhotoData* class and any other managed code data access classes need no awareness of the native code details.

That covers inserting new photos. Returning now to the *PhotoData* class, the remaining methods query by *PhotoId* and stream the selected photo file content from the database into an *Image* object for display. If you've been following along so far, you'll find the rest of our code to be understandable and intuitive since it follows a very similar pattern.

The *SelectPhoto* method accepts a photo ID that is located in the database and returns the string description from the database in an output parameter. The method's return value is a *System.Drawing.Image* object that we will populate with the BLOB streamed in from the database server's NTFS file system using *OpenSqlFilestream*, as shown here:

```
public static Image SelectPhoto(int photoId, out string desc)
{
    const string SelectCmd =
        "SELECT Description, Photo.PathName(), GET_FILESTREAM_TRANSACTION_CONTEXT()" +
        " FROM PhotoAlbum" +
        " WHERE PhotoId = @PhotoId";

    Image photo;

    using(SqlConnection conn = new SqlConnection(ConnStr))
    {
        conn.Open();

        using(SqlTransaction txn = conn.BeginTransaction())
        {
            string filePath;
            byte[] txnToken;

            using(SqlCommand cmd = new SqlCommand(SelectCmd, conn, txn))
            {
                cmd.Parameters.Add("@PhotoId", SqlDbType.Int).Value = photoId;

                using(SqlDataReader rdr = cmd.ExecuteReader(CommandBehavior.SingleRow))
                {
                    rdr.Read();
                    desc = rdr.GetSqlString(0).Value;
                    filePath = rdr.GetSqlString(1).Value;
                    txnToken = rdr.GetSqlBinary(2).Value;
                    rdr.Close();
                }
            }

            photo = LoadPhotoImage(filePath, txnToken);

            txn.Commit();
        }

        conn.Close();
    }

    return photo;
}
```

Once again, we start things off by opening a connection and initiating a transaction. We then execute a simple *SELECT* statement that queries the *PhotoAlbum* table for the record specified by the photo ID and returns the description and full path to the image BLOB, as well as the FILESTREAM transactional context token. And once again we use the path name and transactional context to tie into the server's file system in the *LoadPhotoImage* method, as shown here:

```
private static Image LoadPhotoImage(string filePath, byte[] txnToken)
{
    Image photo;

    SafeFileHandle handle =
        NativeSqlClient.GetSqlFilestreamHandle
         (filePath, NativeSqlClient.DesiredAccess.Read, txnToken);

    using(FileStream fs = new FileStream(handle, FileAccess.Read))
    {
        photo = Image.FromStream(fs);
        fs.Close();
    }

    return photo;
}
```

Just as we did in the *GetOutputFileHandle* method for inserting new photos (only this time using *DesiredAccess.Read* instead of *DesiredAccess.ReadWrite*), we get a *SafeFileHandle* object from our *GetSqlFilestreamHandle* method defined in *NativeSqlClient*. We just saw how this method merely wraps and calls the native SQL client *OpenSqlFilestream* function needed to get the handle for streaming our BLOBs. With this handle, we once again create a new *FileStream*, this time opened for read-only access.

Once we have our *FileStream*, we can deliver the highest streaming performance possible by fire-hosing the BLOB content directly from the NTFS file system on the server into a new *System.Drawing.Image* object by using the static *Image.FromStream* method. The populated image is then passed back up to the form, where it is displayed by using the *Image* property of the *PictureBox* control.

The Payoff

It's time to see all of this in action and give the application a run! To insert a new photo, specify a unique (unused) photo ID, an image file, and a description in the top group box in the PhotoForm window, as shown in Figure 8-8, and then click Save.

FIGURE 8-8 Inserting a new photo into FILESTREAM storage

To select and display the photo and its description back from the database, type its photo ID in the bottom group box, and then click Load. The photo is displayed, as shown in Figure 8-9.

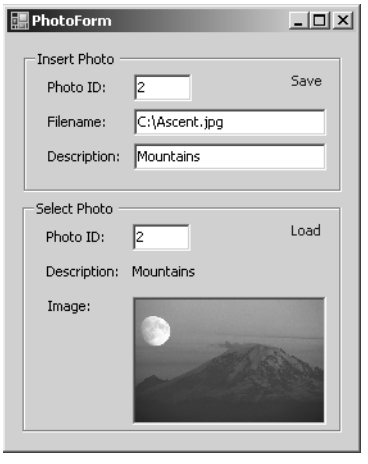

FIGURE 8-9 Retrieving a photo from FILESTREAM storage

This simple application might be small, but it does demonstrate everything needed to leverage the power of FILESTREAM in your .NET client applications. The amount of code required is minimal, and the small amount of code that you do need to write implements fairly straightforward patterns that are easily adapted to various difference scenarios and UIs. For example, in our next FILESTREAM application, we'll stream content from a Hypertext Transfer Protocol (HTTP) service and consume it in a WPF client using the very same FILESTREAM principles that we applied in this Windows Forms application.

Creating a Streaming HTTP Service

We'll now build a simple service as a normal Microsoft ASP.NET Web Application project with a single PhotoService.aspx page. This page can be called by any HTTP client passing in a photo ID value appended to the URL query string; it will stream back the binary content for the specified photo from the database FILESTREAM storage in SQL Server to the client.

To build the service, follow these steps. Start Visual Studio, and then choose File, New, Project. Create a Microsoft Visual C# ASP.NET Web Application project named *PhotoLibraryHttpService* in a solution named *PhotoLibraryFileStreamDemo*, as shown in Figure 8-10.

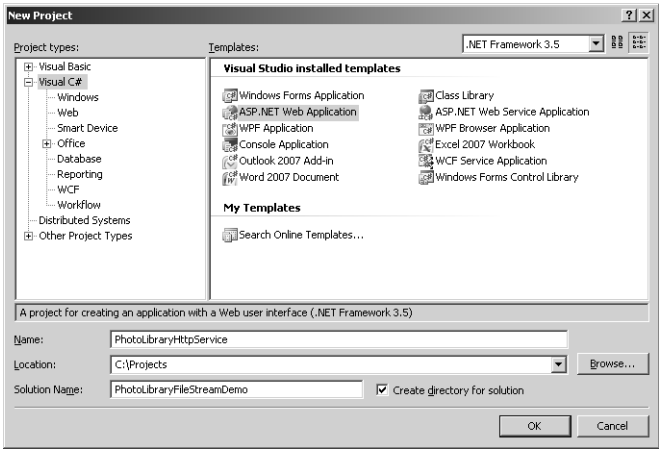

FIGURE 8-10 Creating the streaming HTTP service application

Delete the Default.aspx page created automatically by Visual Studio, and then add a new Web Form named PhotoService.aspx. Unlike a typical .aspx page, this page will not return HTML content. Instead, the page's code-behind class will stream out binary content from the database directly through the *Response* object. For this reason, we delete the HTML markup, leaving only the <@ *Page* %> directive that links the .aspx page with its code-behind class, as shown in Figure 8-11.

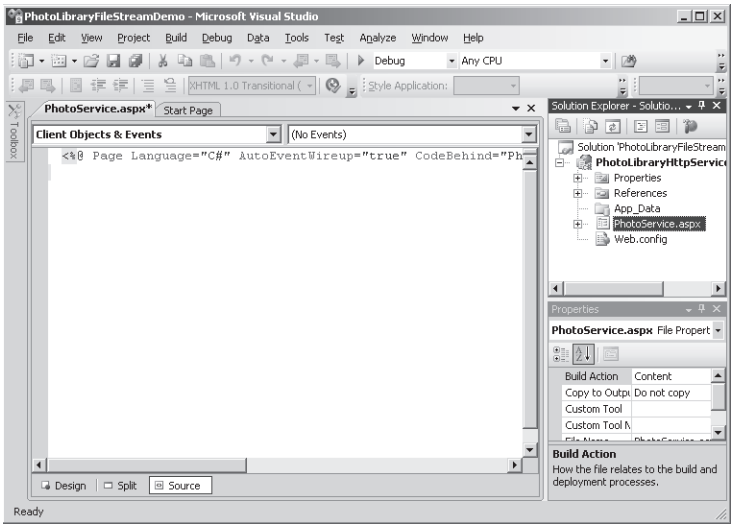

FIGURE 8-11 Creating the PhotoService.aspx page

Make this the default startup page by right-clicking PhotoService.aspx in Solution Explorer and then choosing Set As Start Page. Next, switch to the code-behind class file by right-clicking again on the PhotoService.aspx node in Solution Explorer and then choosing View Code.

Replace the starter code provided by Visual Studio with the code shown in Listing 8-6.

LISTING 8-6 Implementing code for the streaming photo service

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using System.IO;
using Microsoft.Win32.SafeHandles;

namespace PhotoLibraryHttpService
{
  public partial class PhotoService : System.Web.UI.Page
  {
    private const string ConnStr =
        "Data Source=.;Integrated Security=True;Initial Catalog=PhotoLibrary;";

    protected void Page_Load(object sender, EventArgs e)
    {
      int photoId = Convert.ToInt32(Request.QueryString["photoId"]);
      if (photoId == 0)
      {
        return;
      }
```

```
  const string SelectCmd =
      "SELECT Photo.PathName(), GET_FILESTREAM_TRANSACTION_CONTEXT()" +
      " FROM PhotoAlbum" +
      " WHERE PhotoId = @PhotoId";

  using (SqlConnection conn = new SqlConnection(ConnStr))
  {
    conn.Open();

    using (SqlTransaction txn = conn.BeginTransaction())
    {
      string filePath;
      byte[] txnToken;

      using (SqlCommand cmd = new SqlCommand(SelectCmd, conn, txn))
      {
        cmd.Parameters.Add("@PhotoId", SqlDbType.Int).Value = photoId;

        using (SqlDataReader rdr = cmd.ExecuteReader(CommandBehavior.SingleRow))
        {
          rdr.Read();
          filePath = rdr.GetSqlString(0).Value;
          txnToken = rdr.GetSqlBinary(1).Value;
          rdr.Close();
        }
      }

      this.StreamPhotoImage(filePath, txnToken);

      txn.Commit();
    }

    conn.Close();
  }
}

private void StreamPhotoImage(string filePath, byte[] txnToken)
{
  const int BlockSize = 1024 * 512;
  const string JpegContentType = "image/jpeg";

  SafeFileHandle handle =
      NativeSqlClient.GetSqlFilestreamHandle
       (filePath, NativeSqlClient.DesiredAccess.Read, txnToken);

  using (FileStream source = new FileStream(handle, FileAccess.Read))
  {
    byte[] buffer = new byte[BlockSize];
    int bytesRead;
    Response.BufferOutput = false;
    Response.ContentType = JpegContentType;
```

```
            while ((bytesRead = source.Read(buffer, 0, buffer.Length)) > 0)
            {
              Response.OutputStream.Write(buffer, 0, bytesRead);
              Response.Flush();
            }
            source.Close();
          }
        }
      }
    }
```

This code bears a strong resemblance to the code in our earlier Windows Forms application. The *Page_Load* method first retrieves the photo ID passed in via the *photoId* query string value. If no value is passed, the method returns without streaming anything back. Otherwise, as before, the photo file name and transaction context are obtained after establishing a connection and transaction on the database and invoking a *SELECT* statement calling the *PathName* method and the *GET_FILESTREAM_TRANSACTION_CONTEXT* function against the photo ID specified in the *WHERE* clause.

With these two key pieces of information in hand, the *StreamPhotoImage* method is called. The method begins by defining a *BlockSize* integer constant that is set to the reasonable value of 512 KB. As before, picture files larger than this will be streamed to the client in 512-KB pieces. Once again, an indirect call to *OpenSqlFilestream* is made through our *NativeSqlClient* wrapper class to obtain a *SafeFileHandle* that can be used to open an ordinary *FileStream* object and read the BLOB data from SQL Server. This means that you'll also need to create the *NativeSqlClient.cs* class file in this project as you did for the Windows application example (see Listing 8-5). Because this code is executing under the auspices of a Web server, you might also need to grant access to the photo storage directory for the account executing the Web page. This might be *ASPNET* or *NETWORK SERVICE* if you're using Internet Information Services (IIS) or your user account if you're executing the page using Visual Studio's development server.

Note As with all code in this book, the full FILESTREAM demo code in this chapter is available on the book's companion Web site.

Before streaming the binary photo content, we need to change two properties of the *Response* object. In an .aspx page, by default, the *Response* object's *BufferOutput* property is set to *true* and the *ContentType* is set to *text/html*. Here you'll change *BufferOutput* to *false* to deliver optimal streaming performance and inform the client that we're sending a JPEG image by changing the *ContentType* property to *image/jpeg*.

Using a *FileStream* object opened against *SafeFileHandle*, the code then reads from the database FILESTREAM storage in 512-KB chunks and streams to the client using the *Reponse.OutputStream.Write* and *Response.Flush* methods. This is implemented with a simple loop

that reads content from the *FileStream* and sends it to the client via the *Response* object until the entire file is processed.

This completes our service application. Before moving on to build our WPF client, let's first test the service. Press F5 to start the application.

Note Visual Studio may prompt that debugging is not enabled for the application and offer to modify the Web.config file to enable debugging. If this dialog box appears, click OK and allow Visual Studio to modify Web.config for server-side debugging. You might also receive a second dialog box informing you that script debugging is disabled in Internet Explorer. If this dialog box appears, click Yes to continue without client-side debugging enabled.

When Internet Explorer launches PhotoService.aspx, it displays an empty page because no photo ID is present in the URL's query string. In the Address bar, append *?photoId=2* to the URL and reload the page. The code-behind class retrieves photo ID 2 from the database and streams it back for display in the browser, as shown in Figure 8-12.

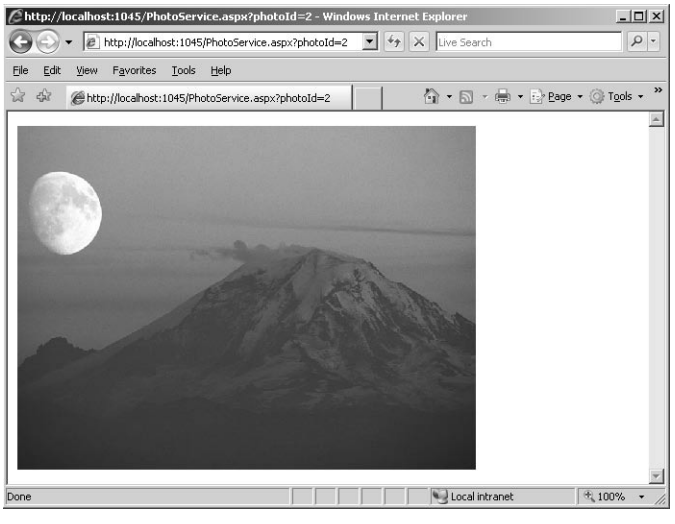

FIGURE 8-12 Streaming a photo over HTTP to Internet Explorer

We've created a functioning HTTP service application that streams pictures from the database to any HTTP client. It's now incredibly easy to build a small WPF client application that calls the service and displays photos. All that's needed is the proper URL with the desired photo ID specified in the query string, as you've just seen. We're using the ASP.NET Development Server provided by Visual Studio, which by default randomly assigns a port number on *localhost* (port 1045 was assigned this time, as indicated in Figure 8-12). We'll need to establish a fixed port number instead so that our WPF client can reliably construct a URL for calling the service. Any unused port number will suffice, so we'll just pick 22111 for this application. To set the port number, right-click the *PhotoLibraryHttpService* project in

Solution Explorer, and then choose Properties. Select the Web tab, select the Specific Port option, and then type **22111** for the port number, as shown in Figure 8-13.

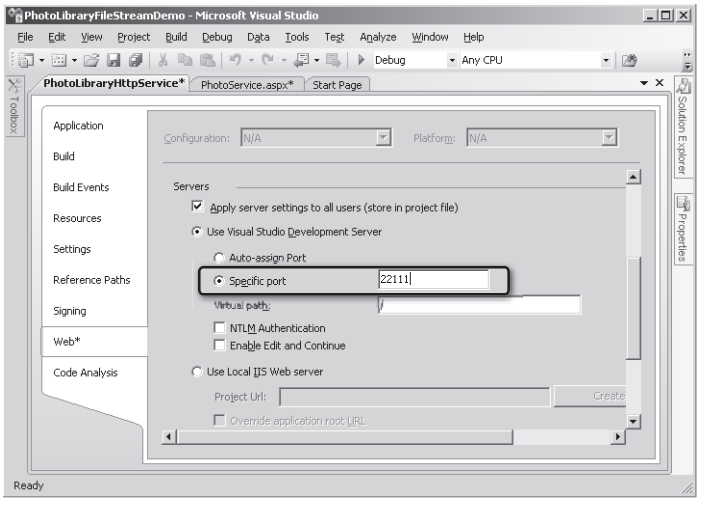

FIGURE 8-13 Setting a specific port number for the HTTP service application

Building the WPF Client

To build the WPF client, follow these steps. In Visual Studio, choose File, New, Project. Create a new Visual C# WPF Application project named *PhotoLibraryWpfClient*. Be sure to select Add To Solution in the Solution drop-down list, as shown in Figure 8-14, so that the project is added to the same *PhotoLibraryFileStreamDemo* solution that contains the *PhotoLibraryHttpService* project.

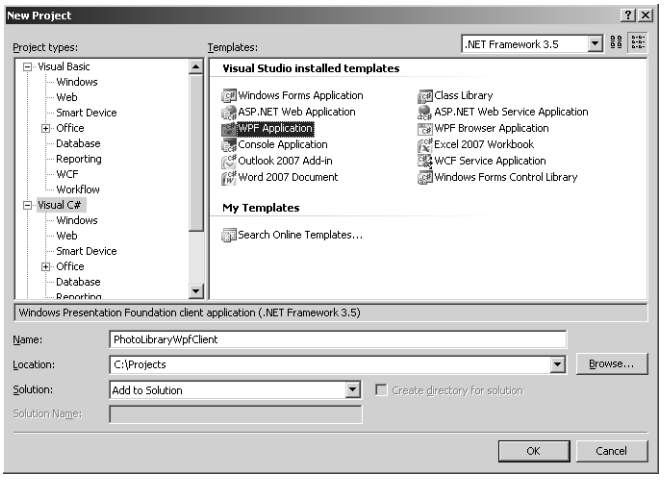

FIGURE 8-14 Creating the streaming WPF client application

Drag *Label*, *TextBox*, *Button*, and *MediaElement* controls from the toolbox, and drop them onto the Window1.xaml design surface. Adjust the control formatting and layout so that the window appears similar to that shown in Figure 8-15.

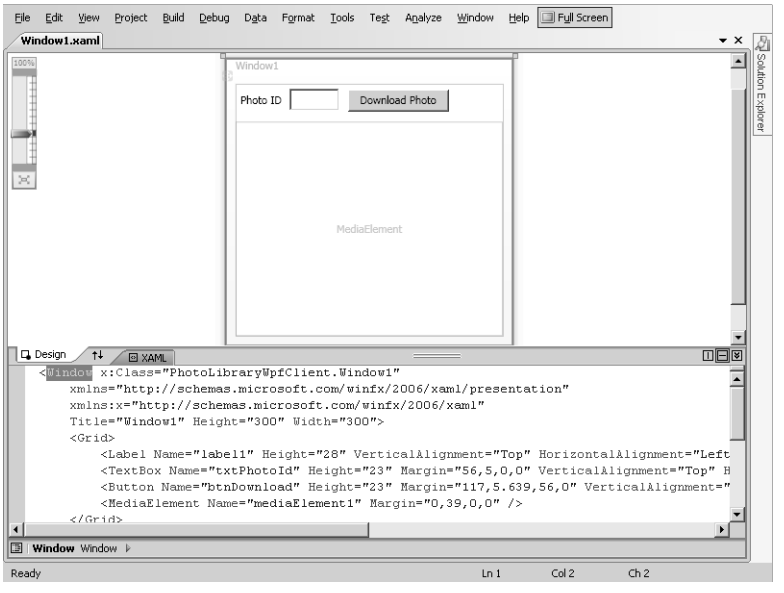

FIGURE 8-15 Simple FILESTREAM WPF UI window

The *MediaElement* control in WPF is a scaled-down media player that is capable of rendering a variety of multimedia types, including images and video, from any source. All we need to do is set its *Source* property to a URL that it can stream its content from. Double-click the *Button* control, and then insert the following code in the button's event handler:

```
private void btnDownload_Click(object sender, RoutedEventArgs e)
{
    string url =
        "http://localhost:22111/PhotoService.aspx?photoId=" +
        this.txtPhotoId.Text;

    this.mediaElement1.Source = new Uri(url);
}
```

This code simply constructs a URL to the PhotoService.aspx page that we know to be running on *localhost* port 22111, passing the desired photo ID in the query string. When the *MediaElement* control's *Source* property is set to that URL, the control automatically calls the service and renders the photo that is streamed from the database to the service and then from the service to the WPF client over HTTP.

To see it work, run the application, and request photo ID 2 for display, as shown in Figure 8-16.

FIGURE 8-16 Streaming a photo from the database over HTTP to a WPF client application

Summary

For applications that work with BLOB data, the new FILESTREAM feature in SQL Server 2008 greatly enhances the storage and performance of unstructured content in the database by leveraging the NTFS file system. It does this while maintaining logical integration between the database and file system that includes transactional support. As a result, we no longer need to make compromises in efficiency and complexity as we did in the past when making the choice between storing BLOB data inside or outside the database.

You also learned how to use the *OpenSqlFilestream* native client API function to deliver high-performance streaming of BLOB content between the file system managed by SQL Server and your Windows, Web, and WPF applications. You can apply the concepts you learned in this chapter across a wide range of applications that require integration between the relational database and a streaming file system.